EXHIBIT C

C1

Page 3 Function Index Fri Jan 04 14:16:53 2008

Page 4 Function Index Fri Jan 04 14:16:53 2008

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMmain.c
**
** Mission Statement: This is the main service file for the EDMsession
                                                                  daemon.
**                         This file contains the main loop,
                                                     and all calls required
**                         to prepare the daemon to go off and service
                                                                     RPC's.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                          USE_SUNRPC - Compile source with sunrpc
                                                                support.  If
**                                    not set, assume DCE support.
**
**                          NONPRODUCTION - Compile source for in house,
                                                                 developer
**                                      testing on local work station.
                                                                   Should
**                                      only be used for targted
                                                                  testing.
**
** Basic idea here: Initalize required locks,
                                              establish signal handlers,
**                   register RPC interface, go wait for RPCs.
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile$ "
                            "$Revision$ "
                            "$Date$" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

/*********************************************************************
                                                                 ********
**
** Routine: main
**
** Inputs: argc, argv
**
** Outputs:  None
**
** Return Codes:
**              exit status
**
** Purpose:     This is the main routine which sets up the daemon
**               to handle RPC calls,
                                 and handles them until it is told
**               to stop or it sees a fatal error.
**
** Intended caller:  None
**
```

```
**********************************************************************
                                                                ********
*/
main (int argc, char *argv[])
{
  /*
  ** Parse options
  */

  (void) parse_commandline(argc, argv);

  /*
  ** Setup logging
  */

  (void) daemon_initialize_logging();

  /*
  ** Enable permanent interrupt catching
  */

  (void) daemon_catch_interrupts();

  /*
  ** Function may not return if improper user running daemon
  */

  (void) daemon_check_proper_ID();

  /*
  ** Function will not return if this fails
  */

  (void) daemon_become_daemon();

  /*
  ** Re-establish log initialization since all "fd's" were
  ** closed by esl_daemon_startup (in daemon_become_daemon)
  */

  (void) daemon_initialize_logging();

  /*
  ** This function doesn't return on failure
  */

  (void) daemon_specific_initialization();

  /*
  ** Unregister service, cleanup cache... Never returns...
  */

  (void) daemon_cleanup();

  /*
  **  Strictly to inhibit compiler warning...
  */

  return( 0 );
}
```

Page 3 of 96 Fri Jan 04 14:16:53 2008

Page 4 of 96 Fri Jan 04 14:16:53 2008

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMRestoreEng.c
**
** Mission Statement: This is the main service file for the EDMsessd
                                                        daemon. This
**                      file contains the callbacks from the main
                                                        function which
**                      prepares the daemon to go off and service RPC's.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                        USE_SUNRPC - Compile source with sunrpc
                                                        support.  If
**                                  not set, assume DCE support.
**
** Basic idea here: Module for UNIX specific daemon initialization
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMsessdd.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdarg.h>
#include <string.h>
#include <syslog.h>
#include <pthread.h>
#include <thread.h>
#include <sys/utsname.h>
#include <netdb.h>

#include <logging/logging.h>
#include <util/esl_core.h>
#include <util/esl_pidfile.h>
#include <util/esl_daemon.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMRestoreEng.h>

#include <EDMmain.h>
#include <EDMRestoreEngLog.h>
#include <EDMProcessManager.h>
#include <EDMProgress.h>
#include <EDMRE_ccr.h>
#include <EDMRE_ccw.h>
#include <EDMRECommandApi.h>
#include <EDMREQuestionApi.h>
#include <EDMREDrainApi.h>
```

```
/*
 * Need to define _XOPEN_SOURCE for signal funtion definitions
 * and certain signal structure definitions.
 */
#define _XOPEN_SOURCE

#include <signal.h>

#undef _XOPEN_SOURCE

static rpc_if_handle_t if_spec;

static int       G_debug = FALSE;
                                /* Variable which will disable forking */

static char      **commandlineargs; /* Pointer to command line args */

/**********************************************************************
                                                              ********
**
** Routine: IsDebugOn
**
** Inputs: None
**
** Outputs:  None
**
** Return Codes:
**               TRUE if debug is on.
**
** Purpose:      This routine can be used to tell other subsystems
**               whether debugging is available.
**
** Intended caller:  internal only.
**
**********************************************************************
                                                              ********
*/
boolean_ty
IsDebugOn()
{
#ifdef DEBUG
    return TRUE;
                        /* if DEBUG defined, we must be in debug mode */
#endif
    if (debugmode)        /* if turned on manually via adb, its on */
        return TRUE;

    return G_debug;
                  /* default is how we were started: -d menas debug */
}
```

```c
/***********************************************************************
                                                                ********
**
** Routine:kill_handler
**
** Inputs: int sigval - the signal which was receieved.
**
** Outputs:  Will log messages telling what action is being taken.
**
** Return Codes:
**              exits with the number of the signal received
**
** Purpose:     This routine handles specific signals i.e. SIGINT,
**              SIGQUIT,
                        SIGTERM. Each results in a log entry and an exit.
**
** Intended caller:  internal only.
**
************************************************************************
                                                                ********
*/
static void kill_handler( IN int sigval )
{
error_status_t   status;
time_t           current_time;
char             *ctimebuf;
char             *ebuff = NULL;

  /* If main exits, it calls this routine with signal 0 */

  /* Unregister the interface */
  (void) csc_unregister_server_interface(&if_spec, &status);

  /* If the unregister fails, report the problem, but continue */
  if ( status != error_status_ok )
  {
    ebuff = (char *) csc_get_error( status );

    (void) EDMRestoreEng_logent(
                         __FILE__,__LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
                            "CSC_SERVER_LOGIN failed: <%d> %s",
                            status, (ebuff ? ebuff : "Unknown error") );
  }

  /* Get the current time */
  (void) time(&current_time);

  ctimebuf = ctime(&current_time);

  /* Overlay newline with null - buf should always be 26 bytes long */
  ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

  (void) EDMRestoreEng_logent(
                       __FILE__,__LINE__, LOG_INFO, MESSAGE_SHUTDOWN, 0,
                       "Shutting down at %s due to signal %d", ctimebuf,
                                                                 sigval);

  exit(sigval);

} /* End of kill_handler() */
```

```c
/***********************************************************************
                                                                ********
**
** Routine: unregister_csc
**
** Inputs: none
**
** Outputs:  Will log messages telling what action is being taken.
**
** Return Codes:
**              none
**
** Purpose:     This routine handles the csc_unregister call
**
** Intended caller:  internal and process manager before exit
**
************************************************************************
                                                                ********
*/
void unregister_csc( void )
{
error_status_t   status;
char     *ebuff = NULL;

  /* Unregister the interface */
  (void) csc_unregister_server_interface(&if_spec, &status);

  /* If the unregister fails, report the problem, but continue */
  if ( status != error_status_ok )
  {
    ebuff = (char *) csc_get_error( status );

    (void) EDMRestoreEng_logent(         __FILE__,__LINE__, LOG_ERR,
                         MESSAGE_CANNOT_UNREGISTER, 0,
                         "CSC_UNREGISTER_SERVER failed: <%d> %s",
                         status, (ebuff ? ebuff : "Unknown error") );
  }

  return;
}
```

```
/*********************************************************************
                                                              ********
 * Function Name:
 *       display_usage
 *
 * Simply displays the usage
 *
 * Call Arguments:
 *       Program name
 *
 * Error Outputs and Side Effects:
 *       Prints usage.
 *
 * Special Considerations:
 *       None.
 *
 *********************************************************************
                                                              ********
 */
static void
display_usage (IN char *progname)
{
  /* Print out usage stmt.  */

  fprintf (stderr,"Usage: %s [-d]\n", progname);
  fprintf (
   stderr,"-d keep the daemon from forking so debugging is easier\n");

} /* end display_usage () */
```

```
/*********************************************************************
                                                              ********
**
** Routine: daemon_catch_interrupts
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     Sets up signals for service. On NT we will have to
**              consider what OS constructs to replace signals with.
**              In this case we are catching SIGTERM, SIGINT, and
**              SIGQUIT and ignoring anything else.
**
** Intended caller:  internal only.
**
*********************************************************************
                                                              ********
*/
void daemon_catch_interrupts()
{
struct sigaction         sactions;       /* Signal actions */

          ZERO( sactions );

        /*
         * Set an empty list so we can set signals we want to handle
         */
         (void) sigemptyset( &sactions.sa_mask );

        /*
         * Add signals that we want to handle
         */
         (void) sigaddset( &sactions.sa_mask, SIGTERM );
         (void) sigaddset( &sactions.sa_mask, SIGINT );
         (void) sigaddset( &sactions.sa_mask, SIGQUIT );

         /* Setup the signal handler. */
         sactions.sa_handler = kill_handler;

        /*
         * Assign handler to each signal we are interested in.
         */
         (void) sigaction( SIGTERM, &sactions, NULL );
         (void) sigaction( SIGINT,  &sactions, NULL );
         (void) sigaction( SIGQUIT, &sactions, NULL );

         /*
         * Setup mask so we can specify what signals we will ignore.
         */
         (void) sigfillset( &sactions.sa_mask );

        /*
         * We want to ignore everything except those we have set up
         * above so remove those from the list.
         */
         (void) sigdelset( &sactions.sa_mask, SIGTERM );
         (void) sigdelset( &sactions.sa_mask, SIGINT );
         (void) sigdelset( &sactions.sa_mask, SIGQUIT );

        /*
        * Set the mask. Since no other threads have been started,
        * all threads will get this mask.
        */
        (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
}
```

```
/*************************************************************************
********
**
** Routine: daemon_check_proper_ID
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              exits with an error when the user is not root
**
** Purpose:     Checks user's ID and determines if the user is allowed
**              to execute service.
                                If there are no constraints then this
**              function may be blank.
**
** Intended caller:  internal only.
**
*************************************************************************
********
*/

void daemon_check_proper_ID()
{
        /*
        ** Check for root
        */

        if (geteuid() != E_ROOTUID)
        {
          (void) EDMRestoreEng_logent(
                  __FILE__,__LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
                          "Must be run as superuser, uid was %d",
                          geteuid());
          exit(1);
        }
}
```

```
/***************************************************************************
                                                                   ********
 **
 ** Routine: parse_commandline
 **
 ** Inputs:      argc, argv (command line arguments)
 **
 ** Outputs:     None
 **
 ** Return Codes:
 **              exits with an error when the user types a bad argument
 **
 ** Purpose:     Parses command line arguments and sets flags. If there
 **              are no flags to be set then this function may be empty.
 **
 ** Intended caller:  internal only.
 **
 ***************************************************************************
                                                                   ********
 */

void parse_commandline(int argc, char *argv[])
{
int                     opt;            /* Process options   */

        commandlineargs = argv;

        while ((opt = getopt(argc,argv,"dD")) != EOF )
        {
          switch(opt)
          {
            case 'd':
            case 'D':
                G_debug = TRUE;
                debugmode = 1;
                                   /* turn on other debugmode flag */
                break;
            default:
                (void) display_usage( argv[0] );
                exit(1);

          }
        }
}
```

```
/***************************************************************************
                                                                   ********
 **
 ** Routine: daemon_initialize_logging
 **
 ** Inputs:      None
 **
 ** Outputs:     None
 **
 ** Return Codes:
 **              None
 **
 ** Purpose:     Do whatever it takes to initialize logging.
                                                               In the near
 **              future this may involve doing something with catalogs
                                                                        or
 **              calling higher level logging functions which
                                                               encapsulate
 **              these things.
 **
 ** Intended caller:  internal only.
 **
 ***************************************************************************
                                                                   ********
 */

void
daemon_initialize_logging()
{
        /* Pass in argv[0], the program name */
        (void) esl_log_init(commandlineargs[0]);
}
```

```
/***********************************************************************
                                                               ********
**
** Routine: daemon_become_daemon
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              exits with an error code if initialization fails
**
** Purpose:     This function is for doing the forking etc. under UNIX.
**              It is unknown what will be necessary under NT.
**
** Intended caller:  internal only.
**
************************************************************************
                                                               ********
*/

void
daemon_become_daemon()
{
         char *ptr;
        int  ret = 0;

        /*
         * Strip the path from the program name so we can use it
         * elsewhere.
         */
        ptr = strrchr(commandlineargs[0], '/');
        if (ptr == NULL)
            ptr = commandlineargs[0];
        else
            ptr++;

        /* Change directory to a process specific core directory */
        ret = esl_coredir_setup(ptr);
        if (ret != 0)
        {
             (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                 MESSAGE_ERR_IN_ESL_COREDIR, errno,
                                 "esl_coredir_setup failed" );
            exit(1);
        }

         /*
         **  This is now esl functionality.
                                  This code does everything necessary
         **  to make this a "real" daemon by detaching from the
                                                              terminal
         **  changing the process group, closing stdout, stderr, stdin,
                                                                   ...
         */

/***    if (G_debug == FALSE)
        {
            ret = esl_daemon_startup();
             if (ret != 0)
            {
                fprintf(
                      stderr, "%s: Failed to initialize as daemon.\n",
                        commandlineargs[0]);
                exit(1);
```

```
            }
        }   **/
}
```

```
/*************************************************************************
                                                                  ********
**
** Routine: rpc_init
**
** Inputs:       None
**
** Outputs:      None
**
** Return Codes:
**               exits with an error code if initialization fails
**
** Purpose:      This function is for doing RPC initialization.
**               For the most part it involves calling the csc routines.
**               This is pretty standard between UNIX and NT.
**
** Intended caller:  internal only.
**
*************************************************************************
                                                                  ********
*/

void rpc_init()
{
error_status_t            status;
                                        /* error status (nbase.h) */
unsigned char    *conn_h;
struct hostent   *hp;
char   *ebuff;
struct utsname name;

/*
** This is here because of HP which may or may not define timeval.
** May be removed when esl_timeval is ported to clients
*/
#ifdef _STRUCT_TIMEVAL
        struct timeval  sleep_interval = {5,0};
                                        /* 5 second sleep interval */
#else
        struct timespec sleep_interval = {5,0};
                                        /* 5 second sleep interval */
#endif

        /* Setup the interface specification for RPC */

        RE_SERVER_IFSPEC(if_spec);

        /*
        * Login as SERVER_PRINCIPAL.  The context of the process
        * will be set to this principal.
        *
        * This process will keep trying to login to DCE if the
                                                         registery
        * server is unavailable.
                           Note that under SUN RPC this is a no-op.
        */
        while (TRUE)
        {
                (void) csc_server_login(RE_SERVER_PRINCIPAL,
                                     RE_SERVER_KEYTAB, &status);

                /* If we succeeded, then exit this loop.  */
                if ( status == error_status_ok )
                {
                        break;
```

```
                }
                else    /* Print error message if appropriate. */
                {
                 ebuff = (char *) csc_get_error( status );

                   (void) EDMRestoreEng_logent(
                                        __FILE__,__LINE__,LOG_ERR,
                                  MESSAGE_NO_LOGIN, 0,
                                  "CSC_SERVER_LOGIN failed: <%d>
                                                            %s",
                                  status, (
                              ebuff ? ebuff : "Unknown error"));
                }

                /* If the failure was due to unavailable client,
                * pause and then try again.
                */
                if (status == sec_rgy_server_unavailable)
                {
                        /*
                         * uses sleep when SUNRPC, otherwise uses
                         * pthread call to delay for the specified
                         * time
                         */
                          CSC_SLEEP(sleep_interval);
                          continue;
                }

                /* If we got here, we had a unexpected failure.  */
                (void) EDMRestoreEng_logent(
                                        __FILE__,__LINE__,LOG_ERR,
                               MESSAGE_NO_LOGIN, 0,
                              "The service cannot log in as
                                                   required");

                exit(1);
        }

        uname(&name);

        hp = gethostbyname(name.nodename);

        if (hp == NULL)
        {
             (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                   MESSAGE_GETHOSTBYNAME_FAIL,
                                                         errno,
                                  "gethostbyname failed" );
             exit(1);
        }

        memcpy((char *) &if_spec.ip_addr, hp -> h_addr, hp->h_length);

         /*
         ** We need to initialize the authorization module before we
                                                                 do
         ** a listen.
         */
         (void)csc_authorization_init(&status);

         if ( status != error_status_ok )
         {
          ebuff = (char *) csc_get_error( status );

            (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                 MESSAGE_NOAUTHORIZATION, 0,
                                 "CSC_AUTHORIZATION_INIT failed: <%d> %s",
                                  status, (
                                        ebuff ? ebuff : "Unknown error") );
            exit(1);
          }

        conn_h = calloc(1, CONNECT_HANDLE_SIZE);

        if (conn_h == NULL)
        {
           (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                  MESSAGE_NO_MEMORY, 0,
                                  "Failure allocating memory for connection
                                                                 handle");
           exit(1);
        }

         (void) csc_register_private_server_interface(
                                                   &if_spec,
                                                 0,
                                                 1,
                                                 conn_h,
                                                 &status);

         if ( status != error_status_ok )
         {
          ebuff = (char *) csc_get_error( status );

           (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                   MESSAGE_CANNOTREGISTER, 0,
                                  "CSC_REGISTER_SERVER_INTERFACE failed:
                                                               <%d> %s",
                                   status, (
                                         ebuff ? ebuff : "Unknown error") );
           exit(1);
         }

         free(conn_h);
}
```

```
/*************************************************************************
                                                                  ********
**
** Routine: rpc_run
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     This function is for running the RPC listen.
**              This is pretty standard between UNIX and NT.
**
** Intended caller:  internal only.
**
*************************************************************************
                                                                 ********
*/

void rpc_run()
{
error_status_t            status;
                                           /* error status (nbase.h) */
char  *ebuff;

         /* listen for RPC calls forever. */
         (void) csc_server_listen(
                             rpc_c_listen_max_calls_default, &status );

        ebuff = (char *) csc_get_error( status );

         /* We don't expect to get here. */
         (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
                                 MESSAGE_SERVERLISTEN, 0,
                                 "CSC_SERVER_LISTEN failed: <%d> %s",
                                  status, (
                                        ebuff ? ebuff : "Unknown error") );
}
```

```
/************************************************************************
                                                                  ********
**
** Routine: daemon_specific_initialization
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     Do whatever makes this daemon special.
                                                        In some cases you
**              may want to start a thread or open a socket.
                                                            Do that here.
**
** Intended caller:  internal only.
**
*************************************************************************
                                                                  ********
*/

void
daemon_specific_initialization()
{
int                         status;
                                                /* error status (nbase.h) */
void                        *statptr;
                                                /* error status (nbase.h) */
int                         ret;
pthread_t                   pmtid;
pthread_t                   progresstid;
pthread_t                   ccrtid;
pthread_t                   ccwtid;
time_t    current_time;
char      *ctimebuf;

          ret = CommandAPIInit(&status);
          ret = QuestionAPIInit(&status);
          ret = DrainAPIInit(&status);

          /* Find out what time it is */
          (void) time(&current_time);

          ctimebuf = ctime(&current_time);

          /* Overlay newline with null - buf should always be 26 bytes
                                                                   long */
          ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

          /* Log startup message */
          (void) EDMRestoreEng_logent( __FILE__, __LINE__, LOG_INFO,
                             MESSAGE_STARTUP, 0,
                                "Restore service %s starting up at %s",
                             commandlineargs[0], ctimebuf );

         /*
          * Start the other threads in the daemon. The main thread
          * becomes the RPC thread. REProcessManager is the
          * entry point for the periodic event thread.
          */
         pthread_create(&pmtid, NULL, REProcessManager, NULL);
         pthread_create(&progresstid, NULL, REProgress, NULL);
         /* pthread_create(&ccrtid, NULL, RestoreSvc_ccr, NULL); */
```

```
          pthread_create(&ccwtid, NULL, RestoreSvc_ccw, NULL);


         rpc_init();
          RestoreSvc_Setup();
         rpc_run();

         pthread_join(pmtid, &statptr);
}
```

```
/************************************************************************
                                                                ********
**
** Routine: daemon_cleanup
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     Call function which will clean up daemon properly.
**
** Intended caller:  internal only.
**
************************************************************************
                                                                ********
*/

void
daemon_cleanup()
{
        kill_handler( 0 );
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMProcessManager.c
**
** Mission Statement: This is the entry point for the Process Manager
                                                              thread.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                          USE_SUNRPC - Compile source with sunrpc
                                                        support.  If
**                                      not set, assume DCE support.
**
** Basic idea here: Module for coding the Process Manager thread.
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMProcessManager.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <syslog.h>
#include <unistd.h>
#include <stdlib.h>

#include <pthread.h>
#include <EDMProcessManager.h>
#include <EDMRECommandApi.h>
#include <EDMRestoreEngLog.h>
#include <EDMmain.h>
#include <restore/restore_engine.h>
#include <restore/restore_api.h>
#include <restore/REprogmsg.h>
#include <restore/EDMREProgressApi.h>
#include <EDMFinalStatus.h>


/* local prototypes */

static void unregister_rpc( void );
static void start_completion( EDMREGlobalStatus );


/* local data */

static boolean_ty        completion_signalled = FALSE;

struct timeout_array
{
```

```
   time_t        timeoutlen;
   time_t        guideadlen;
} tval[MAX_GLOBAL_STATUS_VALUE+1] = {
    // Timeout value        Gui is dead value
  { 5*SECONDS_PER_MINUTE, 5*SECONDS_PER_MINUTE }, // Exiting
  { 5*SECONDS_PER_MINUTE, 5*SECONDS_PER_MINUTE }, // Starting
  { SECONDS_PER_YEAR,     2*SECONDS_PER_HOUR },   // Browsing
  { 3*SECONDS_PER_HOUR,   5*SECONDS_PER_MINUTE }, // Pre phase
  { 10*SECONDS_PER_DAY,   5*SECONDS_PER_MINUTE }, // Execute
  { 2*SECONDS_PER_DAY,    5*SECONDS_PER_MINUTE }  // Post Phase
  };

boolean_ty
IsRestoreTimedOut(IN time_t lastguitime, IN time_t incurrentstate,
                  IN int status)
{
    time_t t = time(NULL);

    if (status > MAX_GLOBAL_STATUS_VALUE)
    {
        return FALSE;
    }

    if (status < 0)
    {
        status = 0;      /* all exiting conditions use same timeout */
    }

    if ( (t - tval[status].guideadlen) > lastguitime)
    {
        return TRUE;
    }
    else if ( (t - tval[status].timeoutlen) > incurrentstate)
    {
        return TRUE;
    }

    return FALSE;
}
```

```
void *
REProcessManager(void *buff)
{
    int  status;
    int  command;
    int  result;
    void         *input_ptr;
    void         *output_ptr;
    boolean_ty   finish_rpc_recvd = FALSE;
    boolean_ty   reader_finish_rcvd = FALSE;
    EDMREGlobalStatus    internal_status;
    time_t       status_time;

    setGlobalStatus( EDMRE_STATE_STARTING );

    while (
!reader_finish_rcvd || !finish_rpc_recvd )   /* until time to exit */
    {
        /* wait for next command */
        if (PopCommand( 1, &command, &status ))
        {
            if (COMMAND_RECORD_GET_FAILED != status)
            /* log error if not 'normal' queue empty error */
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_RECORD_GET_FAILED, 0,
                                      "PopCommand failed;
                                                    status = %d",
                                      status );

            /* check for completion timeout or idle timeout */
            internal_status = getGlobalStatus( &status_time );
            if (completion_signalled)
            {
                if (TRUE == IsRestoreTimedOut(  getLastRpcTime(
                                                ), status_time,
                                             internal_status ) )
                {
                    if (!finish_rpc_recvd)
                    {
                 /* let restore service module clean up, stop rpcs */
                        result = EDMRE_Finish( NULL, NULL );
                        unregister_rpc( );      /* cleanup csc i/f */
                    }
                    EDMRestoreEng_logent(
                                          __FILE__, __LINE__, LOG_ERR,
                              MESSAGE_SHUTDOWN, 0,
                              "Shutting down after timeout awaiting
                                                              sync" );
                    break;
                                           /* escape while to exit */
                }
            } else
            {
                if (TRUE == IsRestoreTimedOut( getLastRpcTime(
                                                ), status_time,
                                             internal_status ) )
                {   /* if already exiting, leave state alone */
                    if (internal_status > 0)
                        start_completion( EDMRE_STATE_TIMEOUT );
                    else
                        start_completion( internal_status );
                }
            }
            continue;
                /* keep waiting in case thread wait got interrupted */
```

```
        }

        /* got some command; see if we're in completion sequence: */
        if (completion_signalled)
        {
            if (COMMAND_FINISH == command && !finish_rpc_recvd)
            {
                if (PopRpcInput( &input_ptr, &status ))
                {
                    EDMRestoreEng_logent(
                                          __FILE__, __LINE__, LOG_ERR,
                                  MESSAGE_POP_RPC_INPUT_FAILED, 0,
                                  "PopRpcInput failed;
                                                status = %d", status );
                }
                else
                    /* let restore service module clean up;
                                                          stop rpc's */
                    result = EDMRE_Finish( input_ptr, &output_ptr );

                finish_rpc_recvd = TRUE;
                unregister_rpc( );
            }
            else if (
           COMMAND_READER_FINISHED == command && !reader_finish_rcvd)
            {
                reader_finish_rcvd = TRUE;
                if (!finish_rpc_recvd) {
                /* let restore service module clean up, stop rpcs */
                    result = EDMRE_Finish( NULL, NULL );
                    unregister_rpc( );  /* cleanup csc i/f */
                }
                break;          /* exit */
            }
            else
            {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                  MESSAGE_INVALID_COMMAND, 0,
                                                      "cmd value: %d",
                                  command );
            }
            continue;
                /* check if both finishes recvd, else keep waiting */
        }

        /* not in completion seq;
                   get pointer to rpc input argument structure */
        if (PopRpcInput( &input_ptr, &status ))
        {
            EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                              MESSAGE_POP_RPC_INPUT_FAILED, 0,
                        "PopRpcInput failed;
                                  status = %d Lost command: %d",
                              status, command );
            continue;   /* ??? keep trying or return ?? */
        }

        switch( command )
        {
        case COMMAND_GET_RESTORABLE_OBJECTS:
            result = EDMRE_GetRestorableObjects(
                                          input_ptr, &output_ptr );
            break;
        case COMMAND_MARK_OBJECT:
            result = EDMRE_MarkObject( input_ptr, &output_ptr );
            break;
            case COMMAND_UNMARK_OBJECT:
                result = EDMRE_UnmarkObject( input_ptr, &output_ptr );
                break;
            case COMMAND_SUBMIT:
                result = EDMRE_Submit( input_ptr, &output_ptr );
                break;

            case COMMAND_START:
                result = EDMRE_Start( input_ptr, &output_ptr );
#if 0       /* taken out to allow continuation after successful & aborted
                                                                  restore */
                start_completion( getGlobalStatus(NULL) );
                                                      /* leave same state */
#endif
                break;

            case COMMAND_FIND_RESTORABLE_OBJECTS:
                result = EDMRE_FindRestorableObjects(
                                                  input_ptr, &output_ptr );
                break;

            case COMMAND_FINISH:
                result = EDMRE_Finish( input_ptr, &output_ptr );
                finish_rpc_recvd = TRUE;
                if (EDMRE_STATE_SUCCESSFUL
                    < (internal_status = getGlobalStatus(
                                                     &status_time ) )
                )
                    /* if already exiting, leave state alone */
                    start_completion( EDMRE_STATE_SUCCESSFUL );
                else
                    start_completion( internal_status );
                unregister_rpc( );
                break;              /* await dispatcher finish command */

            case COMMAND_LOAD_RECX_DIRECTIVES:
                result = EDMRE_Load_recx_directives(
                                                  input_ptr, &output_ptr );
                break;
            case COMMAND_GET_ALL_TIMES:
                result = EDMRE_GetAllBackupTimes(
                                                  input_ptr, &output_ptr );
                break;
            case COMMAND_SET_PREVIOUS_BACKUP:
                result = EDMRE_SetPreviousBackup(
                                                  input_ptr, &output_ptr );
                break;
            case COMMAND_SET_NEXT_BACKUP:
                result = EDMRE_SetNextBackup( input_ptr, &output_ptr );
                break;
            case COMMAND_SET_FIRST_BACKUP:
                result = EDMRE_SetFirstBackup( input_ptr, &output_ptr );
                break;
            case COMMAND_SET_MOST_RECENT_BACKUP:
                result = EDMRE_SetMostRecentBackup(
                                                  input_ptr, &output_ptr );
                break;
            case COMMAND_SET_BACKUP_FOR_TIME:
                result = EDMRE_SetBackupForTime( input_ptr, &output_ptr );
                break;
            default:
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_INVALID_COMMAND, 0,
                                                         "cmd value: %d",
                                      command );
                result = COMMAND_RESULT_FAILURE;
```

```
            }

            /* push result arg structure pointer, IF command succeeded */
            if( result != COMMAND_RESULT_FAILURE)
            {
                if (PushRpcOutput( output_ptr, &status) )
                {
                    EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                          MESSAGE_PUSH_RPC_OUTPUT_FAILED,
                                                                       0,
                                          "PushRpcOutput failed;
                                                           status = %d",
                                          status );
                }
            }

            if (PushResult( result, command, &status) )
            {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                      MESSAGE_FAILURE_TO_QUEUE_RESULT, 0,

                                      "PushResult failed;
                                                  status = %d", status );
            }
        }

#if 0   /* I think we just leave global status as its already set */
        if ( reader_finish_rcvd && finish_rpc_recvd )        /* good exit
                                                                      */
            setGlobalStatus( /* good exit ?? */ 0 );
#endif

        exit ( getGlobalStatus(NULL) );
        return buff;
}
```

```
/* local function to unregister rpc interface */

static void unregister_rpc( void )
{
        sleep( 1 );
                    /* allow last rpc (finish) response to get sent */
        unregister_csc( );      /* stop RPC traffic */
        return;
}
```

```
/* local function to start completion sequence */

static void start_completion( EDMREGlobalStatus status )
{
        setGlobalStatus( status );
        SendFinalStatus( );             /* signal dispatcher */
        completion_signalled = TRUE;
        return;
}
```

```
/**********************************************************************
                                                                ******
**
** File Name:   EDMREProcMgrService.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**         This module contains the Process Manager (
                                        thread) functions that
**         provide the top level processing of the 'asynchronous'
                                                        Restore
**         Engine RPC's.
                      These functions are basically 'wrappers' for the
*          Restore Service Library calls that perform the actual RPC
                                                        services.
**
** Table of Contents:
** -------------------
**                 EDMRE_GetRestorableObjects
**                 EDMRE_MarkObject
**                 EDMRE_UnmarkObject
**                 EDMRE_Submit
**                 EDMRE_Start
**                 EDMRE_Finish
**                 EDMRE_FindRestorableObjects
**
**         Internal Functions:
**                 EDMRE_ProgressCallback
**                 EDMRE_RestoreCallback
**
** Compile-Time Options:
**
**
**********************************************************************
                                                               *****/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

/* #define  _POSIX_SOURCE 1 */


/*
 * System headers.
 */
#include <stdlib.h>
#include <sys/syslog.h>
```

```
#include <unistd.h>
#include <string.h>

/*
 * Epoch headers.
 */

#include <ebutil/ebutil.h>
#include <restore/restore_engine.h>
#include <restore/REprogmsg.h>
#include <restore/EDMREProgressApi.h>

/*
 * Local headers
 */
#include <RSLapi.h>
#include <EDMProcessManager.h>
#include <EDMRestoreEngLog.h>
#include <EDMRECommandApi.h>


/*
 * Local functions
 */
static boolean_ty EDMRE_ProgressCallback( unsigned long progress );
static boolean_ty EDMRE_RestoreCallback( void );

/**********************************************************************
                                                              ********
**
** Routine:  EDMRE_GetAllBackupTimes
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                           arg struct
**
** Return Codes:
**            0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
**********************************************************************
                                                              ********
*/
int EDMRE_GetAllBackupTimes( void *input_args, void **output_args )
{
    RE_get_all_backup_times_args          *in_args
        = (RE_get_all_backup_times_args *)input_args;
    RE_get_all_backup_times_result        *out_args;

    int           status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_get_all_backup_times_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                    __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_get_all_backup_times_args" );
        status = COMMAND_RESULT_FAILURE;         /* fatal error */
    }

    else
    {
          out_args->cookie = in_args->cookie;
          out_args->status = RSTSL_GetAllBackupTimes(
                                                     in_args->startTime,
                                                       in_args->endTime,

                                                    in_args->maxEntries,
                                                       in_args->flags,

                                                &out_args->backupTimes,

                                                 &out_args->numEntries,

                                                     &out_args->cookie);
          *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_get_all_backup_times_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_GetRestorableObjects
**
** Inputs:   void *input_args     ptr to struct with RPC input args
**
** Outputs:  void **status        addr of void * to receive ptr output
                                                              arg struct
**
** Return Codes:
**             0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_GetRestorableObjects( void *input_args, void **output_args )
{
    RE_get_restorable_objects_start_args         *in_args
         = (RE_get_restorable_objects_start_args *)input_args;
    RE_get_restorable_objects_output_result      *out_args;

    int           status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(
                          RE_get_restorable_objects_output_result) );
    if (NULL == out_args)
    {
         EDMRestoreEng_logent(
                       __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                 0,
          "calloc fail for RE_get_restorable_objects_output_result" );
         status = COMMAND_RESULT_FAILURE;         /* fatal error */
    }

    else
    {
         out_args->cookie = in_args->cookie;
         out_args->status = RSTSL_GetRestorableObjects(
         (restorableObjectPtr)in_args->parentObj->RE_restorable_obj_u.
                                                               tloInfo,
                                        in_args->parentObj->objLevel,
                                        &out_args->childrenObjs,
                                        &out_args->cookie,
                                        in_args->maxEntries,
                                        &out_args->numEntries,
                                        in_args->allowBadFiles );
         *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_get_restorable_objects_start_args, (
                                                      char *)in_args);
    free( in_args);

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_MarkObject
**
** Inputs:   void *input_args     ptr to struct with RPC input args
**
** Outputs:  void **status        addr of void * to receive ptr output
                                                              arg struct
**
** Return Codes:
**            0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_MarkObject( void *input_args, void **output_args )
{
    RE_mark_object_args           *in_args = (
                                       RE_mark_object_args *)input_args;
    RE_get_mark_results_result    *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_get_mark_results_result) );
    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                      __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_get_mark_results_result" );
        status = COMMAND_RESULT_FAILURE;         /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_MarkObject( in_args->thisObj,
                                             in_args->backupTime,
                                             in_args->allowBadFiles,
                                             in_args->descend,
                                             &out_args->badFileCount,

                                        &out_args->permDenyFileCount,
                                             &out_args->fileMarkCount,
                                             &out_args->dirMarkCount,

                                           &out_args->otherMarkCount,
                                             EDMRE_ProgressCallback );
        *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_mark_object_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_UnmarkObject
**
** Inputs:   void *input_args     ptr to struct with RPC input args
**
** Outputs:  void **status        addr of void * to receive ptr output
                                                              arg struct
**
** Return Codes:
**            0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_UnmarkObject( void *input_args, void **output_args )
{
    RE_unmark_object_args          *in_args = (
                                     RE_unmark_object_args *)input_args;
    RE_get_unmark_results_result *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_get_unmark_results_result) );
    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                      __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_get_unmark_results_result" );
        status = COMMAND_RESULT_FAILURE;         /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_UnmarkObject( in_args->thisObj,
                                               in_args->backupTime,
                                               in_args->badFilesOnly,
                                               in_args->descend,
                                               &out_args->badFileCount,
                                               &out_args->fileMarkCount,
                                               &out_args->dirMarkCount,

                                             &out_args->otherMarkCount,
                                               EDMRE_ProgressCallback );
        *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_unmark_object_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine: EDMRE_ProgressCallback
**
** Inputs:   unsigned long progress      objects processed so far
**
** Outputs:  none
**
** Return Codes:
**           boolean_ty          FALSE if operation can continue
**                               TRUE if operation should be cancelled
**
** Purpose:  Restore service library callback function to be called
                                                          periodically
**             to return progress information and check for
                                                         cancellation.
**
***********************************************************************
                                                                ********
*/
static boolean_ty EDMRE_ProgressCallback( unsigned long progress )
{
        UpdateProgressValue( progress );
        return TestRpcCancelFlag( );
}
```

```
/***********************************************************************
                                                                ********
**
** Routine: EDMRE_RestoreCallback
**
** Inputs:   none
**
** Outputs:  none
**
** Return Codes:
**           boolean_ty          FALSE if operation can continue
**                               TRUE if operation should be cancelled
**
** Purpose:  Restore service library callback function to be called
                                                          periodically
**           by 'Start' function to return check for cancellation.
**
***********************************************************************
                                                                ********
*/
static boolean_ty EDMRE_RestoreCallback( void )
{
        EDMREGlobalStatus       internal_status;
        long    last_rpc_time;
        time_t  status_time;

        internal_status = getGlobalStatus( &status_time );
        last_rpc_time = getLastRpcTime( );

        if (internal_status == EDMRE_STATE_USER_QUIT
         || internal_status == EDMRE_STATE_ADMIN_QUIT)   /* someone
                                                           aborted */
                return TRUE;
        if (TRUE == IsRestoreTimedOut( last_rpc_time, status_time,
                                       internal_status )
        )
                return TRUE;
                                               /* no sign of user life */
        return TestRpcCancelFlag( );
                                              /* user-signalled cancel */
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  EDMRE_Submit
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                           arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**           asynchronously from main RPC thread
**
**********************************************************************
                                                             ********
*/
int EDMRE_Submit( void *input_args, void **output_args )
{
    RE_submit_args                  *in_args = (
                                        RE_submit_args *)input_args;
    RE_get_submit_results_output          *out_args;
    unsigned int                   object_count = 0;
    EDMRST_submit_args            *submitArgs = calloc(1,sizeof(
                                            EDMRST_submit_args));
    int            status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_get_submit_results_output) );
    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                    __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_get_submit_results_output" );
        status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
    {
        submitArgs->clientSocketPort = in_args->socketPort;
         submitArgs->mapfile_env = esl_strdup(in_args->mapFile_env);
         submitArgs->socketClientNm = esl_strdup(
                                   in_args->socketClientName);

         out_args->submitObjectID = in_args->submitObjectID;
         out_args->status = RSTSL_Submit( in_args->hostname,
                                  in_args->overwritePolicy,
                                  in_args->inPlace,
                                  in_args->directory,
                                  in_args->transport,
                                  &out_args->submitObjectID,
                                  &object_count,
                                  EDMRE_ProgressCallback,
                                  submitArgs);
         out_args->objectsDone = object_count;
         *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_submit_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  EDMRE_Start
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                           arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**           asynchronously from main RPC thread
**
**********************************************************************
                                                             ********
*/
int EDMRE_Start( void *input_args, void **output_args )
{
    RE_start_args         *in_args = (RE_start_args *)input_args;
    RE_status_result      *out_args;

    int            status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof(RE_status_result) );
    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                    __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_status_result" );
        status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_Start( in_args->submitObjectID,
                                    EDMRE_RestoreCallback );
        *output_args = (void *)out_args;
    }

    xdr_free( xdr_RE_start_args, (char *)in_args);
    free( in_args );

    return status;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  EDMRE_Finish
**
** Inputs:   void *input_args      OPTIONAL ptr to struct with RPC input
                                                                  args
**
** Outputs:  void **status         OPTIONAL addr of void * to receive
                                                                ptr to
**                                 output arg struct
**
** Return Codes:
**          0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**           asynchronously from main RPC thread
**
*********************************************************************
                                                            ********
*/
int EDMRE_Finish( void *input_args, void **output_args )
{
        int      status = COMMAND_RESULT_SUCCESS;
        RE_status_result         *out_args;

        out_args = calloc( 1, sizeof(RE_status_result) );

        if ( (out_args->status = RSTSL_Finish( )) != E_SUCCESS)
                status  = COMMAND_RESULT_FAILURE;

        if (NULL != input_args)
        {
                xdr_free( xdr_RE_null_args, (char *)input_args);
                free( input_args );
                *output_args = (void *)out_args;
        }
        else    /* only keep output struct if user want it */
                free( out_args );

        return status;
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  EDMRE_FindRestorableObjects
**
** Inputs:   void *input_args     ptr to struct with RPC input args
**
** Outputs:  void **status        addr of void * to receive ptr output
                                                         arg struct
**
** Return Codes:
**          0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**           asynchronously from main RPC thread
**
*********************************************************************
                                                            ********
*/
int EDMRE_FindRestorableObjects(
                                void *input_args, void **output_args )
{
        RE_find_restorable_objects_args         *in_args =
                        (RE_find_restorable_objects_args *)input_args;
        RE_find_restorable_objects_result       *out_args;
        EBREC_SearchCriteriaRec
                                                       searchCriteria;

        int     status = COMMAND_RESULT_SUCCESS;

        out_args = calloc( 1, sizeof(
                        RE_find_restorable_objects_result) );
        if (NULL == out_args)
        {
                EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_MEMORY, 0,
                        "calloc fail for
                                RE_find_restorable_objects_result" );
                status = COMMAND_RESULT_FAILURE;
                                                    /* fatal error */
        }

        else
        {
                /* prepare search criteria structure for input to
                                                      RSTST func: */
                strncpy( searchCriteria.startDirectory,
                        in_args->searchCriteria->startDirectory,
                                                              256);
                searchCriteria.descendDirectory =

                          in_args->searchCriteria->descendDirectory;
                strncpy( searchCriteria.searchString,
                        in_args->searchCriteria->searchString, 128);
                searchCriteria.excludeString =

                             in_args->searchCriteria->excludeString;
                searchCriteria.typeOfFile =
                                in_args->searchCriteria->typeOfFile;
                strncpy( searchCriteria.owner,
                        in_args->searchCriteria->owner, 64);
                searchCriteria.excludeOwner =
                                in_args->searchCriteria->excludeOwner;
                strncpy( searchCriteria.group,
                        in_args->searchCriteria->group, 64);
```

```
                    searchCriteria.excludeGroup =
                              in_args->searchCriteria->excludeGroup;
                    searchCriteria.sizeInBytes.high =
                              in_args->searchCriteria->sizeInBytes.
                                                                high;
                    searchCriteria.sizeInBytes.low =
                              in_args->searchCriteria->sizeInBytes.
                                                                 low;
                    searchCriteria.sizeMatch =
                              in_args->searchCriteria->sizeMatch;
                    searchCriteria.startTime =
                              in_args->searchCriteria->startTime;
                    searchCriteria.endTime =
                              in_args->searchCriteria->endTime;

                    out_args->status = RSTSL_FindRestorableObjects(
                                                        &searchCriteria,

                                                   EDMRE_ProgressCallback );
                    *output_args = (void *)out_args;
          }

          xdr_free( xdr_RE_find_restorable_objects_args, (
                                                 char *)in_args );
          free( in_args );

          return status;
}
```

```
/***********************************************************************
                                                                 ********
**
** Routine:  int EDMRE_Load_recx_directives
**
** Inputs:  RE_recx_file_info *fileinfo   Information on file to be
                                                              retreived
**
** Outputs: Error or success from the RSTSL call
**
** Return Codes:
**            0 for success and non-zero for failure.
**
** Purpose:  Function to retrieve directives file from client and then
**           load the file contents into the context structure.
                                                               The file
**           transfer is done with edm link.
**
***********************************************************************
                                                                ********
*/
int EDMRE_Load_recx_directives( void *input_args,
                                void **output_args )
{
  RSTRPC_recx_file_info * fileinfo = (
                                 RSTRPC_recx_file_info *)input_args;
  RE_status_result *outargs;
  outargs = calloc(1,sizeof(RE_status_result));

  /*
   * Actually load the recx structure.
   */
  outargs->status = RSTSL_Load_recx_directives(fileinfo);
  *output_args = (void *)outargs;

  /*
   * Return that the RPC was atleast successfule,
                                            the load may not have been
   */
  return COMMAND_RESULT_SUCCESS;

}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_SetPreviousBackup
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                              arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_SetPreviousBackup( void *input_args, void **output_args )
{
    RE_set_backup_time_args      *in_args
        = (RE_set_backup_time_args *)input_args;
    RE_get_all_backup_times_result       *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_status_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                     __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_status_result" );
        status = COMMAND_RESULT_FAILURE;          /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_SetPrevBackup( in_args->flags);
        *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_SetBackupForTime
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                              arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_SetBackupForTime( void *input_args, void **output_args )
{
    RE_backup_for_time_args      *in_args
        = (RE_backup_for_time_args *)input_args;
    RE_get_all_backup_times_result       *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_status_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                     __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_status_result" );
        status = COMMAND_RESULT_FAILURE;          /* fatal error */
    }
    else
    {
        out_args->status = RSTSL_SetBackupForTime( in_args->time,
                                                   in_args->flags);
        *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_backup_for_time_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  EDMRE_SetNextBackup
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                           arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
*********************************************************************
                                                             ********
*/
int EDMRE_SetNextBackup( void *input_args, void **output_args )
{
    RE_set_backup_time_args       *in_args
        = (RE_set_backup_time_args *)input_args;
    RE_get_all_backup_times_result        *out_args;

    int           status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_status_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                   __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_status_result" );
        status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_SetNextBackup( in_args->flags);
        *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  EDMRE_SetFirstBackup
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                           arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
*********************************************************************
                                                             ********
*/
int EDMRE_SetFirstBackup( void *input_args, void **output_args )
{
    RE_set_backup_time_args       *in_args
        = (RE_set_backup_time_args *)input_args;
    RE_get_all_backup_times_result        *out_args;

    int           status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_status_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                   __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_status_result" );
        status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
    {
        out_args->status = RSTSL_SetFirstBackup( in_args->flags);
        *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  EDMRE_SetMostRecentBackup
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output
                                                          arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
************************************************************************
                                                                ********
*/
int EDMRE_SetMostRecentBackup( void *input_args, void **output_args )
{
    RE_set_backup_time_args      *in_args
         = (RE_set_backup_time_args *)input_args;
    RE_get_all_backup_times_result       *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_status_result) );


    if (NULL == out_args)
    {
         EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                 0, "calloc fail for RE_status_result" );
         status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
    {
         out_args->status = RSTSL_SetMostRecentBackup( in_args->flags);
         *output_args = (void *) out_args;
    }

    xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
    free( in_args);

    return status;
}
```

```
/*
** ====================================================================
                                                              ========
** Copyright 1996,1997 EMC Corporation
** ====================================================================
                                                              ========
*/
/*
** ====================================================================
                                                              ========
** EDMFinalStatus.c
**
** Mission Statement:
**
**
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                    support.  If
**                                  not set, assume DCE support.
**
** Basic idea here: Module for Control Channel Reader thread.
** ====================================================================
                                                              ========
*/
/*
** ====================================================================
                                                              ========
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ====================================================================
                                                              ========
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMFinalStatus.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <csc/cscomm.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif

#include <EDMutils.h>
#include <restore/dispatch_daemon.h>
#include <restore/csc_Dispatch_Protocol_Client.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol.h>
#include <EDMFinalStatus.h>

#ifdef __cplusplus
}
#endif
#include <EDMRE_ccr.h>
#include <EDMSession.h>
#include <logging/logging.h>
#include <EDMReturnMessageApi.h>
#include <EDMRestoreEngLog.h>

// Global/Extern if_spec to be used by RE_ccw/DD_ccr.
rpc_binding_handle_t *restoreService_ccw_handle_p;
extern DD_client_session_id *p_restoreServiceUid;

int
SendFinalStatus(void)
{
 int lrc=0;
 int status=0;

         lrc = PushResponseMessage( (int) dp_final_stats_indicate,
                                    *p_restoreServiceUid,
                                    restoreService_ccw_handle_p,
                                    &status );

 return(lrc);
}
```

```
/**********************************************************************
                                                                 ******
**
** File Name:   RSLinitfin.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**                This module contains the Restore Service Library
                                                          functions to
**                initialize and terminate the restore operation.
**
** Table of Contents:
** -------------------
**                RSTSL_Initialize
**                RSTSL_Finish
**
**        Internal Functions:
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
**********************************************************************
                                                                *****/

/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif

/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1

/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <dlfcn.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLstartup.h>

/*
 * #defines, structures, typedefs local to this source file
 */

static eerrno_ty init_plugins( restore_context *rcp );
static int validate_plugin( struct pluginData *piDataPtr );

/*
 * External declarations
 *
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;

/*
 * Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                          These must be in the same order and position
 * as the PIFuncIndex values defined in RSLplugin.h.
 */

char *piFuncNames[PIFuncIndexLast+1] = {
        "RSTPI_Identify",
        "RSTPI_Initialize",
        "RSTPI_GetTopLevelObjects",
        "RSTPI_SetTopLevelObject",
        "RSTPI_GetNextLevelObjects",
        "RSTPI_ClearRestoreContext",
        "RSTPI_Submit",
        "RSTPI_GetTopLevelTemplates",
        "RSTPI_DoesAlternateExist",
        "RSTPI_MarkObject",
        "RSTPI_UnmarkObject",
        "RSTPI_IsObjectMarked",
        "RSTPI_IsObjectMarkable",
        "RSTPI_GetAllBackupTimes",
        "RSTPI_GetCurrentBackupTime",
        "RSTPI_SetBackupForTime",
        "RSTPI_SetPrevBackup",
        "RSTPI_SetNextBackup",
        "RSTPI_SetFirstBackup",
        "RSTPI_SetMostRecentBackup",
        "RSTPI_IsTherePrevBackup",
        "RSTPI_IsThereNextBackup",
        "RSTPI_IsTherePrevBackupForTime",
        "RSTPI_IsThereNextBackupForTime",
        "RSTPI_Finish",
        "RSTPI_StartRestore",
        "RSTPI_FindRestorableObjects",
        "RSTPI_GetFindResults",
        "RSTPI_GetNecessaryMedia"
};
```

```
/*********************************************************************
 * RSTSL_Initialize:
 *
 *  This function takes care of all the initialization for a restore
 *  session.  This must be called prior to any of the other functions
 *  in the Restore API.
 *
 * Parameters:
 *
 *  userName (I) - The name of the user.
 *
 *********************************************************************/

eerrno_ty
RSTSL_Initialize( const char *userName )
{
    eerrno_ty status = E_SUCCESS;

    /*
     * If we have not yet allocated space for a restore_context
     * structure, do so now.  If we have already done so,
                                                        just clear it
     * now.
     */

    if (NULL == rcp)
    {
        rcp = (struct restore_context *)malloc(sizeof(
                                         struct restore_context));
        if (NULL == rcp)
         {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
            return(EP_RB_RECOVER_NOMEM);
         }
    }
    memset(rcp, 0, sizeof(struct restore_context));

    rcp->rc_human_uidname = esl_strdup( userName );


    if (!rcp->rc_human_uidname) {
      rec_api_log_csm(SUB_CSM_NOMEM, NULL);
      return(EP_RB_RECOVER_NOMEM);
    }

    /*
     * Set the appropriate field in the recovery context to indicate
     * that this recover session is based on the Recover API.
     * This flag is in place for historical reasons but is used by
     * other parts of the Recover API library.
     */

    rcp->gui_mode = 1;

    /*
     * Initialize the logging mechanism.
     */

    if (status = rbrlog_begin(rcp, progname))
    {
        return(status);
    }

    /*
     * Initialize the few "recover context" variables that we can at
     * this early stage.
```

```
     */

    setup_proc(rcp);

    /*
     * The following call will:
     *   -Initialize the saveset datatbase.
     *   -Infer any information we can at this point.
     */

    if (status = startup(rcp))
    {
        return(status);
    }

    /* Do plugins setup: Find and initialize all valid restore plugin
                                                                libs: */

    status = init_plugins( rcp );

    return( status );
}       /* End of RSTSL_Initialize() */
```

```
/*********************************************************************
 * RSTSL_Finish
 *
 *    Function Description:
 *
 *  This function terminates a restoral session,
                                        but not while a restore is in
 *  progress.
         It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session.
 *
 * Parameters:
 *
 *  none
 *
 */

eerrno_ty
RSTSL_Finish( void )
{
    int mc_n;


    eerrno_ty err = E_SUCCESS;

    if (NULL == rcp)
    {
        return( E_SUCCESS );
    }
    RemoveSubmitFiles();
    /*
     * Call rbr_cleanup() which kills the aux proc(
                                                s), unlocks the work
     * item, then calls rbrlog_end(
                                    ) to enter the last logs and to close
     * the log file.
     */

    rbr_cleanup(rcp);

    /*
     * Deallocate the memory of restore_context and the related
     * structures.
     */

    if (NULL != rcp->rc_mcp)     /* Free the multicat structures */
    {
        mcat_destroy(rcp->rc_mcp);
    }

    /*
     * Free the mark bit map space
     */

    for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
    {
        if (NULL != rcp->rc_marks[mc_n])
        {
            free(rcp->rc_marks[mc_n]);
        }
        rcp->rc_marks[mc_n] = NULL;
    }

    if (NULL != rcp->rc_nmarks_by_plane)
```

```
    {
        free(rcp->rc_nmarks_by_plane);
    }

    /*
     * Free the configuration structures
     */

#if 0
    if (NULL != rcp->rc_cfgname)
    {
        free(rcp->rc_cfgname);
    }
#endif

    if (NULL != rcp->rc_config)
    {
        rbc_freeconfig(rcp->rc_config);
    }

    /*
     * Free the DS_NONE structures array
     * Note that even though rc_dsnones is the head of linked list
     * of dsnone_info structures, the list is allocated via malloc
     * as an array initially (ref. alloc_plane_arrays()), therefore
     * we can do a free here.
     */

    if (NULL != rcp->rc_dsnones)
    {
        free(rcp->rc_dsnones);
    }

    /*
     * Free the volume list structures.
     */

    if (NULL != rcp->ebvllist)
    {
        (void)ebvl_volidlist_destructor(
                                        rcp->ebvllist, EBVL_DESTROY_ALL);
    }

    /*
     * Free the plugin related data
     */

    rcp->rc_backup_app = 0;
    while (rcp->currentPIptr = rcp->piList)
    {
        rcp->rc_backup_app++;
        rcp->appData = rcp->currentPIptr->appData;
        /* allow plugin to clean up and close .so: */
        if ( E_SUCCESS != (err =
            rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
                                                               rcp) ) )
        {
        /* log error, continue */
            rbe_user_error( err,
                        "RSTPI_Finish failed for restore plug-in
                                                      library %s\n",
                ((struct pluginIDdata *)(
                        rcp-> currentPIptr-> idData))-> name );
        }
        dlclose( rcp-> currentPIptr-> libHdl );
        rcp->piList = rcp->piList->next;
```

```
            free (rcp->currentPIptr);
    }

    /*
     * Free the various simple string buffers
     */

    if (NULL != rcp->rc_top_level_object_name)
    {
          free(rcp->rc_top_level_object_name);
    }

    if (NULL != rcp->rc_template_name)
    {
          free(rcp->rc_template_name);
    }

    if (NULL != rcp->rc_workitem_name)
    {
          free(rcp->rc_workitem_name);
    }

    if (NULL != rcp->rc_human_uidname)
    {
          free(rcp->rc_human_uidname);
    }

    if (NULL != rcp->rc_effective_uidname)
    {
          /* don't free, its internal: free(rcp->rc_effective_uidname);
                                                                      */

    }

    if (NULL != rcp->rc_client_rbuname)
    {
          free(rcp->rc_client_rbuname);
    }

    if (NULL != rcp->rc_client_hostname)
    {
          free(rcp->rc_client_hostname);
    }

    if (NULL != rcp->rc_client_scriptname)
    {
          /* don't free, its internal: free(rcp->rc_client_scriptname);
                                                                      */

    }

    if (NULL != rcp->rc_client_dirtop)
    {
          free(rcp->rc_client_dirtop);
    }

    if (NULL != rcp->rc_cmd_context)
    {
          /* don't free -- its internal/temp data: free(
                                          rcp->rc_cmd_context); */
    }

    if (NULL != rcp->rc_source_client_hostname)
    {
          free(rcp->rc_source_client_hostname);
    }

    if (NULL != rcp->rc_cpiogen_executable)
```

```
    {
          /* don't free, its internal: free(rcp->rc_cpiogen_executable);
                                                                       */

    }

    if (NULL != rcp->rc_plugin_wi_types)
    {
          free(rcp->rc_plugin_wi_types);
    }

    if (NULL != rcp->rc_pwd)
    {
          free(rcp->rc_pwd);
    }

    /*
     * Finally, deallocate the restore_context itself
     */

    free(rcp);
    rcp = NULL;



    return( err );
}         /* RSTSL_Finish */
```

```
/***************************************************************************
 * init_plugins
 *
 *    Function Description:
 *
 *  This function locates, opens,
                          validates and initializes all restore
 *  plug-in (shared) libraries.  They must be located in
 *  /usr/epoch/EB/cure_plugin (
                              eb_cure_plugin_dir).  All .so files in that
 *  directory are opened and validates for version# and presence of
                                                                  all
 *  mandatory functions.
                             The RSTPI_Identify function is called for each
 *  library to determine which optional features are supported,
                                                              and that
 *  the corresponding functions are present.  Finally,
                                                     the RSTPI_Initialize
 *  function is called for each valid library.
 *
 * Parameters:
 *
 * Inputs:
 *       rcp      (I)      - Pointer to restore context
 *
 * Outputs:
 *       none
 *
 * Returns:
 *       E_SUCCESS or EP_RB_RECOVER_xxx
 *
 * Logic/pseudo code:
 *
 *       open plugin dir
 *       while read_next_entry succeeds
 *               verify .so file (else continue)
 *               open shared library file (else continue)
 *               on errors below:
 *                       close shared library file
 *                       continue
 *               fetch all mandatory function addresses
 *               call Identify function
 *               validate version number
 *               fetch all indicated optional function addrs
 *               call Initialize function
 *               add workitem types to composite exclusion list
 *               add to valid plugin list
 *       close plugin dir
 */

static eerrno_ty **init_plugins**( restore_context *rcp )
{
        DIR             *dirp;
        struct dirent   *direntp;
        eerrno_ty       status = E_SUCCESS;
        struct pluginData *piDataPtr = NULL;
        struct pluginData *piListPtr = NULL;
        int             val_result;
        struct pluginIDdata *idDataPtr;
        char            *tmp_types;
        int             shlib_dirlen;
        char            shlib_path [MAXPATHLEN];
```

```
        /* open plugin directory or bust */
        if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
        {
                rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
#if 1
                return E_SUCCESS;
                                        /* allow continuation w/o plugins */
#else
                return EP_RB_RECOVER_NO_PLUGINS;
                                                        /* later do this */
#endif
        }

        strcpy( shlib_path, eb_cure_plugin_dir );
        strcat( shlib_path, "/" );
        shlib_dirlen = strlen (shlib_path);

        /* loop thru entries in directory*/
        while (NULL != (direntp = readdir( dirp )))
        {
                if (NULL == piDataPtr)
                {       /* allocate next plugin data structure */
                        if (NULL == (piDataPtr
                                   = calloc( 1, sizeof(
                                             struct pluginData) )))
                        {
                                status = EP_RB_RECOVER_NOMEM;
                                break;
                                               /* fall thru to cleanup */
                        }
                }

                if (NULL == strstr( direntp->d_name, ".so" ) )
                        continue;       /* skip this guy */

                strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
                if (NULL == (piDataPtr->libHdl
                           = dlopen( shlib_path, RTLD_NOW )))
                {
                        rbe_user_error( 0,
                            "Error opening restore plug-in library
                                                             %s: %s\n",
                                        direntp->d_name, dlerror() );
                        continue;       /* skip this one */
                }

        /* Fetch addresses of all mandatory functions and */
        /* Do Identify processing: call it, save options, validate */

                if (0 != (val_result = validate_plugin(
                                                        piDataPtr ) ))
                {
                    if (val_result == -1 || val_result == -4)
                    {
                        rbe_user_error( 0,
                "Functions missing from restore plug-in library %s:
                                                                  %s\n",
                                        direntp->d_name, dlerror() );
                    }
                    else if (val_result < 0)
                    {
                        rbe_user_error( 0,
                          "Validation failed for restore plug-in
                                                        library %s\n",
                                    direntp->d_name );
                    }
```

```
                        else
                        {
                                rbe_user_error( val_result,
                "RSTPI_Identify failed for restore plug-in library
                                                                %s\n",
                                                direntp->d_name );
                        }

                        dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
                        piDataPtr->libHdl = NULL;
                        continue;   /* on any error, skip this lib */
                }

        /* let DC plug-in do its initialization */
                rcp->appData = NULL;
                                /* enter plugin with clean appdata */
                status =
                piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
                if (E_SUCCESS != status)
                {
                        rbe_user_error( status,
                "RSTPI_Initialize failed for restore plug-in library
                                                                %s\n",
                                                direntp->d_name );
                        dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
                        piDataPtr->libHdl = NULL;
                        status = E_SUCCESS;
                                                /* this was't fatal */
                        continue;
                                        /* on any error, skip this lib */
                }

        /* save plugin's appData */
                piDataPtr->appData = rcp->appData;
                rcp->appData = NULL;

        /* add piDataPtr to valid plugin list */
                if (NULL == piListPtr)
                        rcp->piList= piDataPtr;
                                                /* first in list */
                else
                        piListPtr->next = piDataPtr;
                                                /* link from prev */
                piListPtr = piDataPtr;
                                                /* new end of list */
                piDataPtr = NULL;

        /* add workitem types to composite exclusion list */
                idDataPtr = (struct pluginIDdata *)piListPtr->idData;
                if (idDataPtr->num_types > 0)
                {
                    tmp_types = calloc( 1, 1 + idDataPtr->num_types
                                    + rcp->rc_num_plugin_wi_types
                                                                    );
                    if (NULL == tmp_types) {
                        status = EP_RB_RECOVER_NOMEM;
                        break;
                    }
                    if (NULL != rcp->rc_plugin_wi_types)
                    {
                  /* move old list to new buffer and free old list */
                        memcpy( tmp_types,
                                rcp->rc_plugin_wi_types,
                                rcp->rc_num_plugin_wi_types );
```

```
                        free( rcp->rc_plugin_wi_types );
                    }
                    memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
                            idDataPtr->wi_types,
                            idDataPtr->num_types );
                    rcp->rc_num_plugin_wi_types +=
                                            idDataPtr->num_types;
                    tmp_types[rcp->rc_num_plugin_wi_types] = 0;
                    rcp->rc_plugin_wi_types = tmp_types;
                }
        }

        (void)closedir( dirp );

        /* free up leftovers: */
        if (NULL != piDataPtr)
                free (piDataPtr);

        if (E_SUCCESS != status)
        { /* Free contents of plugin list: */
                while (NULL != (piDataPtr = piListPtr))
                { /* allow plugin to clean up and close .so: */
                        rcp->appData = piDataPtr->appData;
                        piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                                rcp );
                        dlclose( piDataPtr->libHdl );
                        piListPtr = piDataPtr->next;
                        free (piDataPtr);
                }
        }

        return status;
}
```

```
        /* init_plugins */

/***********************************************************************
 * validate_plugin
 *
 *   Function Description:
 *
 *  This function retrieves the addresses of the mandatory plugin
                                                             functions
 *  and stores them in the function pointer array.
                                          If any function is missing
 *  it returns -1.
         It then calls the identify function and verifies wthe plugin
 *  version,
          and finds its optional functions. Specific error values are
 *  returned on version mismatch and missing optional functions.
 *
 * Parameters:
 *
 * Inputs:
 *       piDataPtr (
              I)   - Pointer to plugin data structure with libHdl set
 *
 * Outputs:
 *       piFuncArray in piDataPtr is loaded with pointers to plugin
                                                             functions
 *
 * Returns:
 *       0 on success
 *       -1 on any missing required functions
 *       -2 if version validation fails OR identify returns junk
 *       -3 if workitem type validation fails
 *       -4 on any missing optional functions indicated by options
                                                                 flags
 *       +n (
   EB_RB_RECOVER_xxx) for error codes returned from Identify function
 *
***********************************************************************
                                                                  ****/

static int validate_plugin( struct pluginData *piDataPtr )
{
        int             index;
        eerrno_ty       status;
        struct pluginIDdata *idDataPtr;

        for( index = 0; index <= PIFuncIndexLastBasic; index++ )
        {
                if (NULL == (piDataPtr->piFuncArray[index]
                           = (piFuncPtr) dlsym( piDataPtr->libHdl,
                                                piFuncNames[index]
                                                                  ))
                )
                        return -1;
        }

        /* call identify and validate: */
        status = piDataPtr->piFuncArray[PIFuncIndexIdentify](

                                                &piDataPtr->idData );
        if (status != E_SUCCESS)
                return status;
        if (NULL == (idDataPtr = (
                        struct pluginIDdata *)piDataPtr->idData) )
                return -2;
```

```
        if (idDataPtr->version != RSTPI_VERSION)
        { /* only version 1 supported so far */
                piDataPtr->idData = NULL;
                return -2;
        }

        if (idDataPtr->num_types && !idDataPtr->wi_types)
        {       /* count cant be positive with null pointer */
                piDataPtr->idData = NULL;
                return -3;
        }

        /* if startRestore option set, get its addr or bust */
        if ( ( (RSTPI_OPTION_SPECIAL_START
                == (idDataPtr->options & RSTPI_OPTION_MASK_START))
            && (NULL == (
                      piDataPtr->piFuncArray[PIFuncIndexStartRestore]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                             piFuncNames[PIFuncIndexStartRestore] )))
             )
        /* OR if special find option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_FIND
                == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
            && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFind]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                   piFuncNames[PIFuncIndexFind] )))
               || (NULL == (
                       piDataPtr->piFuncArray[PIFuncIndexFindResults]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                               piFuncNames[PIFuncIndexFindResults])
             ) ) )        )
        /* OR if special getMedia option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
                == (
                   idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
            && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                               piFuncNames[PIFuncIndexGetMedia] )))
        )     )
        {
                piDataPtr->idData = NULL;
                return -4;
        }

        return 0;
}
```

```
/*      validate_plugin */
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/


/* EDMReturnMessageApi.cc
 *
 *
 * Mission Statement:  file that contains an API to manage the Message
                                                                Queues
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 *                   A few calls to manage the Message Queues.
 */

#if !defined(lint)
static char     RCS_id [] = "@(
                                  #)$RCSfile: EDMReturnMessageApi.cc,v $ "
                                "$Revision: 1.0 $ "
                                "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

#include <logging/logging.h>
#include <csc/cscomm.h>
#include <eerrno/e_eb.h>


// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/gqueue.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_daemon.h>

#ifdef __cplusplus
}
#endif

#include <EDMSession.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMReturnMessage.h>
#include <EDMReturnMessageApi.h>

declare(RWGQueue, EDMReturnMessage)

RWGQueue(EDMReturnMessage) g_messageQueue;

static pthread_mutex_t  G_returnmessagemutex =
                                         PTHREAD_MUTEX_INITIALIZER;
```

```
/************************************************************************
                                                                ********

**
** Routine:  LockReturnMessageMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the return message object
**
************************************************************************
                                                                ********
*/

static void
LockReturnMessageMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_returnmessagemutex, NULL);
    }

    pthread_mutex_lock(&G_returnmessagemutex);
}

/************************************************************************
                                                                ********

**
** Routine:  UnlockReturnMessageMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the return message object
**
************************************************************************
                                                                ********
*/

static void
UnlockReturnMessageMutex()
{
    pthread_mutex_unlock(&G_returnmessagemutex);
}

/************************************************************************
                                                                ********

**
** Routine:  PushResponseMessage
**
** Inputs:   int msgtoqueue - an integer representing the message to
                                                                queue.
**           DD_client_session_id sid - Source service of message.
**
```

```
** Outputs:  int *status - a status for the push command
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Push a message on the Message Queue
**
*********************************************************************
                                                             ********
*/

int
PushResponseMessage(IN int msgtoqueue,
                     IN DD_client_session_id sid,
                    IN rpc_binding_handle_t *csc_h,
                    OUT int *status)
{
    EDMReturnMessage *msg, *ret;

    if (status == NULL)
    {
        return 1;
    }

    // At one point we did check the csc_h (
                                   CSC handle) but we found that
    // we would rather take a NULL and check it once we try to use it.
    // That way we can handle cases where the private service messages
                                                                   us
    // and we haven't made a connection to it yet.

    if (msgtoqueue == 0)
    {
        *status = RETURNMESSAGE_BAD_PARAM;
        return 1;
    }

    //
    // Acquire a new message Object.
    //
    msg = new EDMReturnMessage();
    if (msg == NULL)
    {
        *status = RETURNMESSAGE_NO_MEMORY;
        return(1);
    }

    //
    // Build the message
    //
    msg -> setSessionID(sid);
    msg -> setMessage(msgtoqueue);
    msg -> setTimeIssued(time(NULL));
    msg -> setBindingHandle(csc_h);

    LockReturnMessageMutex();

    ret = g_messageQueue.append(msg);

    UnlockReturnMessageMutex();

    if (ret == NULL)
    {
        *status = RETURNMESSAGE_QUEUE_APPEND_FAILURE;
        delete msg;
        return 1;
```

```
    }

    return 0;
}

/*********************************************************************
                                                             ********
**
** Routine:  PopResponseMessage
**
** Inputs:
**
** Outputs:  int *status - a status for the pop message
**           int *msgid   - a place to put the message to queue
**           DD_client_session_id *sess - a place to put the
**                       session ID
**            rpc_binding_handle_t **client_h_p - a handle to respond
                                                                  on.
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Pop a message off the Message Queue.
                                                  If the Queue is empty
**           block until a signal wakes us up. If the timeout value is
**           0 the thread blocks until a message is received,
                                                            otherwise
**           it will block the length of time specified.
**
*********************************************************************
                                                             ********
*/

int
PopResponseMessage(
              OUT int *ResponseMessage, OUT DD_client_session_id *sess,
                  OUT rpc_binding_handle_t **client_h_p,
                                                     OUT int *status)
{
    EDMReturnMessage *ret;

    if (status == NULL)
    {
        return 1;
    }

    *status = 0;

    if (client_h_p == NULL || ResponseMessage == NULL || sess == NULL)
    {
        *status = RETURNMESSAGE_BAD_PARAM;
        return -1;
    }

    LockReturnMessageMutex();

    if (g_messageQueue.isEmpty())
    {
        *status = RETURNMESSAGE_QUEUE_IS_EMPTY;
        UnlockReturnMessageMutex();
         return -1;
    }

    ret = g_messageQueue.get();

    UnlockReturnMessageMutex();
    if (ret == NULL)
    {
        *status = RETURNMESSAGE_RECORD_GET_FAILED;
        return -1;
    }

    ret -> getSessionID(sess);
    *ResponseMessage = ret -> getMessage();
    *client_h_p = ret -> getBindingHandle();

    delete ret;

    return 0;
}
```

```
/*********************************************************************
                                                               ******
**
** File Name:   EDMDispProtocolSvc.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**      This module contains the callback functions for use with
**       the dispatch daemon protocol.
**
**Table of Contents:
** ------------------
**
**
** Compile-Time Options:
**             This section must list any compile time definitions
**             which will affect this header.
**
** Copyright (c) 1996 by EMC Corp.
**
*********************************************************************
                                                               *****/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispProtocolSvc.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1999/03/06 09:00:00 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>
#include <util/esl_string.h>

#include <logging/logging.h>
#include <csc/cscomm.h>
#include <eerrno/e_eb.h>

#include <sys/time.h>
#include <pthread.h>
#include <EDMccw.h>
#include <EDMDD_ddp.h>
#include <EDMutils.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/csc_Dispatch_Protocol_Service.h>

#include <EDMDispatchSession.h>
#include <EDMTimedMessageApi.h>
#include <EDMReturnMessageApi.h>
#include <EDMDDHandleMgrApi.h>

#include <logging/logging.h>
#include <EDMDispatchLog.h>

#ifdef __cplusplus
}
#endif

/*********************************************************************
```

```
                                                               ********
**
**  Structures
**
*********************************************************************
                                                               ********
*/

/*********************************************************************
                                                               ********
**
** Routine:  dp_connect_indicate_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
*********************************************************************
                                                               ********
*/
int *
dp_connect_indicate_1_svc(
                      DP_connect_indicate_msg *msg,struct svc_req *req)
{
 int rc;
 int status;
 rpc_binding_handle_t *client_handle_p = NULL;

  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
  {
     EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                      "UpdateSessionLastReceived failed.");
  }

  /* Get the csc_binding_handle associated with this sid */
  rc = GetCSCHandle(&msg->sid,
                    &client_handle_p,
                    &status );
  if (0 != rc)
  {
     EDMDispatch_logent(
       __FILE__,__LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                      "GetCSCHandle failed.");
  }

  /* Push message to send onto the queue */
  rc = PushResponseMessage((int) dp_connect_confirm,
                           msg->sid,
                           client_handle_p,
                          &status);
  if (0 != rc)
   {
     EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                      "PushResponseMessage failed.");
   }
   if (IsDebugOn())
   {
       (void) EDMDispatch_logent(
                     __FILE__,__LINE__, LOG_ERR, DDP_SENDING_MESSAGE, 0,
                          "Pushing response message to restore service.
                                                                     ");
   }

 return((int*)0);
}
```

```
/*************************************************************************
                                                                 ********
**
** Routine:  dp_abort_response_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
*************************************************************************
                                                                 ********
*/
int *
dp_abort_response_1_svc(
                       DP_abort_response_msg *msg,struct svc_req *req)
{
 int rc;
 int status;


  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
   {
      EDMDispatch_logent(
         __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                          "UpdateSessionLastReceived failed.");
   }

  /* Remove the timed message to indicate that we got the response */
  rc = deleteTimedMessage(&msg->sid,
                          dp_abort_request,
                          &status);
  if (0 != rc)
   {
      EDMDispatch_logent(
     __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                          "deleteTimedMessage failed.");
   }

 return( (int*)0 );
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  dp_close_response_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
**********************************************************************
                                                               ********
*/

int *
dp_close_response_1_svc(
                       DP_close_response_msg *msg,struct svc_req *req)
{
 int rc;
 int status;

  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
   {
     EDMDispatch_logent(
       __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                         "UpdateSessionLastReceived failed.");
   }

  /* Remove the timed message to indicate that we got the response */
  rc = deleteTimedMessage(&msg->sid,
                          dp_close_request,
                          &status);
  if (0 != rc)
   {
     EDMDispatch_logent(
    __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                         "deleteTimedMessage failed.");
   }

 return( (int*)0 );
}
```

```
/**********************************************************************
                                                               ********
**
** Routine:  dp_ping_response_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
**********************************************************************
                                                               ********
*/

int *
dp_ping_response_1_svc( DP_ping_response_msg *msg,struct svc_req *req)
{
 int rc;
 int status;

  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
   {
     EDMDispatch_logent(
       __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                         "UpdateSessionLastReceived failed.");
   }

  /* Remove the timed message to indicate that we got the response */
  rc = deleteTimedMessage(&msg->sid,
                          dp_ping_request,
                          &status);
  if (0 != rc)
   {
     EDMDispatch_logent(
    __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
                         "deleteTimedMessage failed.");
   }

 return( (int*)0 );
}
```

```
/***********************************************************************
                                                               ********
**
** Routine:  dp_event_indicate_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
************************************************************************
                                                               ********
*/

int *
dp_event_indicate_1_svc(
                          DP_event_indicate_msg *msg,struct svc_req *req)
{
 int rc;
 int status;
 rpc_binding_handle_t *client_handle_p;

  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
   {
      EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                          "UpdateSessionLastReceived failed.");
   }

 return( (int*)0 );
}
```

```
/***********************************************************************
                                                               ********
**
** Routine:  dp_progress_indicate_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
************************************************************************
                                                               ********
*/

int *
dp_progress_indicate_1_svc(
                    DP_progress_indicate_msg *msg,struct svc_req *req)
{
 int rc;
 int status;

 /* Update last time we herd from the service */
 rc = UpdateSessionLastReceived( &msg->sid );
 if (0 != rc)
   {
      EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                          "UpdateSessionLastReceived failed.");
   }

 return( (int*)0 );
}
```

```
/**********************************************************************
                                                              ********
**
** Routine:  dp_final_stats_indicate()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
**********************************************************************
                                                              ********
*/

int *
dp_final_stats_indicate_1_svc(DP_final_stats_indicate_msg *msg,
                              struct svc_req *req)
{
 int rc;
 int status;
 rpc_binding_handle_t *client_handle_p = NULL;

 /* Update last time we herd from the service */
 rc = UpdateSessionLastReceived( &msg->sid );
 if (0 != rc)
 {
     EDMDispatch_logent(
       __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                       "UpdateSessionLastReceived failed.");
 }
 /* Get the csc_binding_handle associated with this sid */
 rc = GetCSCHandle(&msg->sid,
                   &client_handle_p,
                   &status );
 if (0 != rc)
 {
     EDMDispatch_logent(
      __FILE__,__LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                       "GetCSCHandle failed.");
 }

 /* Push message to send onto the queue */
 rc = PushResponseMessage(dp_final_stats_confirm,
                          msg->sid,
                          client_handle_p,
                          &status);
 if (0 != rc)
 {
     EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                       "PushResponseMessage failed.");
 }

 return( (int*)0 );
}
```